|epcc|

User's Guide to the EPCC's BlueSky Service
Version 1.0

October 22, 2007

Contents

1 Introduction **4**

2 Architecture Overview **4**
2.1 Processors, Caches and Memory 4
2.2 Operational modes of the machine 4
2.3 Building blocks of the system 5
2.4 Networks 5
2.5 Front-end nodes 6

3 Getting started **6**
3.1 Web Administration Account 6
3.2 Machine accounts 7
3.3 Logging in 7
3.4 Password changes 7
3.5 System maintenance 7
3.6 Transferring files 7
3.7 Editors 8

4 User Resource Management **8**
4.1 Diskspace 8
4.2 The BlueSky Administration Web Site 8
4.3 Charging 8

5 Compilation **9**
5.1 Compilers 9
5.1.1 Fortran compiler 9
5.1.2 C compiler 9
5.1.3 C++ compiler 9
5.1.4 GNU compilers 9
5.2 Compiling MPI code 9
5.2.1 Fortran code 10
5.2.2 C code 10
5.2.3 C++ code 10
5.3 Useful compiler options 10
5.4 Example Makefiles 10
5.4.1 MPI fixed format Fortran code 10
5.4.2 MPI Fortran 90 code 11

5.4.3 MPI C code . . . 12

6 File Management **13**
6.1 Use of `/tmp` . . . 13
6.2 Backup policies . . . 13

7 Porting Codes **13**
7.1 Timers . . . 13
7.2 Default sizes . . . 14
7.2.1 FORTRAN codes . . . 14
7.2.2 C/C++ codes . . . 14
7.3 MPI Datatypes . . . 15
7.3.1 Fortran . . . 15
7.3.2 C . . . 15

8 Batch Processing **15**
8.1 Sample Co-processor mode job submission file . . . 16
8.2 Sample Virtual Node mode job submission file . . . 17
8.3 Submitting and monitoring your job . . . 17
8.4 Multistep Submission . . . 18
8.5 More on mpirun . . . 19
8.5.1 Running in Virtual-node mode or Co-processor mode . . . 19
8.5.2 Process Mapping . . . 19
8.5.3 Output from your job . . . 19
8.5.4 Environment Variables . . . 19
8.5.5 Further options of `mpirun` . . . 20
8.6 Accounts linked to more than one project . . . 20

9 Tools **20**
9.1 Debugging with gdb . . . 20
9.2 Traceback . . . 21
9.3 MPI trace . . . 22
9.3.1 Function Trace . . . 22
9.4 MP_Profiler . . . 23
9.4.1 Profiling . . . 23
9.4.2 Tracing . . . 23
9.5 TAU . . . 24
9.6 KOJAK . . . 24
9.6.1 Hardware Counters with KOJAK . . . 25
9.7 Xprofiler . . . 26
9.8 HPM . . . 26
9.9 Paraver . . . 27

10 Libraries **28**
10.1 BLAS Library . . . 28
10.2 FFTW 2.1.5 . . . 29
10.2.1 Optimised FFTW . . . 29
10.2.2 Standard FFTW . . . 29
10.3 Global Arrays . . . 29
10.4 ESSL . . . 30

11 Optimisation **30**
11.1 Compiler flags . . . 30
11.2 16-byte alignment of data . . . 31

12 Applications **33**

13 Support, Training and Documentation **33**
13.1 The BlueSky Administration Web Site 33
13.2 The Helpdesk 34
13.3 Training 34
13.4 Documentation 34

14 References and Further Reading **34**
14.1 BlueSky service documentation: 34
14.2 IBM Product Documentation: 34
14.3 Useful papers and slides (IBM): 34
14.4 Third-party Documentation 35

1 Introduction

Welcome to the EPCC's BlueSky service.

Important note: The BlueSky service is new and many things are still to be installed. Also this document is still under development and we expect things to change. Please check back regularly for changes.

2 Architecture Overview

The BlueSky system offers a total of 1024 compute chips (nodes) in a single cabinet. Each chip has two processors, which makes a total of 2048 processors. The theoretical peak performance of the system is 5.7 TFlops.

2.1 Processors, Caches and Memory

A BlueGene chip (node) has two standard PowerPC 440 processing cores. The processors have a clock frequency of 700 MHz. This clock frequency is relatively low by today's standards, but is one of the key ingredients for the low power design, allowing for the very large number of processors inside a single cabinet.

Each processing core has a specially designed double floating point unit, known as *Hummer*^{2}. These units consist of two standard PowerPC FPUs. Each double unit can complete one 64-bit FMA (fused-multiply-add) en each side per cycle. This equates to four Flops per cycle and a peak performance of 2.8 GFlops per processor. The two sides of the Hummer^2 are not independent however. IBM calls their operation mode SIMOMD (single instruction multiple operation multiple data). In addition to SIMD operations (both sides perform the same operation on different data), there are a number of special instructions that perform different operations in the two sides. This feature can be exploited to complete a complex FMA every two cycles. For the efficient use of the double FPU 16-byte alignment of the data is required, see section 11.2.

The Level 1 (L1) cache is part of the PowerPC 440 core. Each core has 32 kB of L1 data cache and 32 kB of L1 instruction cache. They are 64-way set associative and feature a round robin replacement policy. The cache line size is 32 bytes. Note that the L1 caches are *not* coherent between the two cores of a chip. This does not pose any problems to MPI codes.

Each processor has a small Level 2 (L2) cache of 2 kB. This is smaller than the L1 and acts as a prefetch buffer only. The cache line size is 128 bytes. The L2 cache is fully associative. Coherency of the two L2 caches is achieved by snooping.

On each chip there is one Level 3 cache of 4 MB. This is made from embedded DRAM and is shared between the two processors of the chip. The L3 cache is organised as two 8-way set associative banks. The latency of the L3 is approximately 35 cycles.

The BlueSky service is configured with 512 MB of DDR memory per chip, shared between the two processing cores. The memory is directly mapped to the application's address space. There is no TLB. The memory latency is approximately 85 cycles.

2.2 Operational modes of the machine

The BlueGene chips offer a number of operational modes. The two most important modes are *Co-processor mode* (CPM) and *Virtual node mode* (VNM). In the Co-processor mode all computations are

	Processors	Chips/Nodes	Compute Cards	Main Memory	I/O Nodes
Chip/Node:	2	1	—	512 MB	—
Compute Card:	4	2	1	1 GB	—
Pset:	16	8	n/a	4 GB	1
Node Bord:	64	32	16	16 GB	4
Midplane:	1024	512	256	256 GB	64
Cabinet/Rack:	2048	1024	512	512 GB	128

Table 1: Assets associated with the individual building blocks

performed on the first processor of each chip. The second processor is use for communications. CPM has the advantage that computations and communications can be overlapped.

In the Virtual node mode, both processors of a chip form independent virtual nodes and each processor runs its own MPI task. The resources of the node (chip) are split between the two processors, e.g. 256 MB of main memory per task. The two processors share access to the L3 cache and the communication networks. Computations and communications can not be overlapped in VNM. The advantage of VNM is a higher peak performance of 5.6 GFlops per chip compared to 2.8 GFlops in CPM.

2.3 Building blocks of the system

To build the entire BlueSky service from the chips, these are grouped into several intermediate building blocks, which have certain assets attached to them. Each chip forms an individual *compute node*. Two chips are attached to a single *compute card*. Sixteen compute cards are placed on a *node board*. The BlueSky cabinet is filled with thirty two node boards. The cabinet can be divided into two *midplanes*, which is the smallest partition with a closed torus network in all three dimensions, (see section 2.4 for details of the networks). The building blocks and their assets are summarised in Table 1.

In addition to the compute nodes, there are dedicated *I/O nodes*. The BlueSky service is configured with one I/O node per eight compute nodes. The compute nodes run a lightweight Linux derived compute node kernel (CNK), which offers only very limited functionality. The I/O nodes run a full Linux kernel, which includes some custom extensions. The rationale is to keep the compute nodes as uninterrupted by the operating system as possible, by outsourcing the operating system tasks to dedicated extra hardware. The machine is divided into 128 *Psets*. Each Pset consists of eight compute nodes and one I/O node. Within each Pset the compute nodes communicate their I/O requirements to the I/O node via the global tree network. The I/O node uses the Gigabit ethernet network to communicate with the "outside world", for example the file server, the front-end nodes and the service node. The networks are discussed in Section 2.4.

2.4 Networks

The BlueSky architecture has five different networks:

- Three dimensional torus
- Global tree
- Global barrier and interrupts
- Gigabit Ethernet
- Control Network

For point-to-point communications the torus network is used. This network is connected to all compute nodes. The processors are arranged on a simple grid with six nearest neighbours (up, down,

right, left, front & back). The entire machine is configured as an $8 \times 8 \times 16$ processor grid with periodic boundary conditions in all three dimensions (3D-torus). I/O nodes are not connected to the torus network.

The global tree network is used for the fast implementation of global operations such as reductions and broadcasts. This network has several arithmetic operations build into it. Maximum and minimum can be performed for integer and floating point numbers. Summations and logical operations are available for integer numbers. The global tree is also used to connect the compute nodes with their respective I/O node.

The global barrier and interrupts network is also organised as a tree. It offers a significantly reduced latency compared to the global tree network, at the expense of a reduced functionality.

The Gigabit ethernet network is used to communicate with the file server, the front-end and service node. This is only active in the I/O nodes. Compute nodes use the global tree network to off-load their I/O tasks to the I/O nodes.

Finally there is the control network, which is connected to the JTAG of all compute and I/O nodes. This network is used for booting the system and monitoring its health and performance. Again this is an ethernet network.

2.5 Front-end nodes

The user interface to the BlueSky service is provided by an IBM eServer BladeCenter JS20 with 4 blades, which are known as the front-end nodes. The front-end nodes are used for editing, compilation and job submission. They run SUSE Linux.

3 Getting started

3.1 Web Administration Account

All users of the service must first register via the BlueSky web site. To do this, users will need the project code and password provided by their project Principal Investigator (PI) or Project Manager (PM). If the PI does not have these, the **PI** or **PM** needs to contact the EPCC Helpdesk. The email address is:

`epcc-support@ed.ac.uk`

Once you have the project code and password, go to the BlueSky web site at:

`https://www.epcc.ed.ac.uk/safadmin`

and follow the instructions for new user registration. Users will be asked to provide a preferred e-mail address which will act as a unique identifier for BlueSky administration. All communication with the user will be via this e-mail address, so it is essential that it is kept up-to-date. This is the user's responsibility.

Once the registration form is submitted, both the user and the project PI will receive an e-mail to that effect. Once confirmation of the registration from the PI has been received, an administration account will be created. The user will then receive a further e-mail with details of the account and initial password.

If encountering a delay between registration and receipt of account details, please check first with the project PI: it is their responsibility to confirm the registration. The process should usually take 1–2 working days.

Please report any problems with the web-site to:

`epcc-support@ed.ac.uk`

See section 4.2 and section 13.1 for more details of the web-site.

3.2 Machine accounts

It is important to realise that users will need to register for a web administration account before receiving a login account and password for the service machine. The former is used to access administration services via the web site, while the latter is a normal machine login account. The two should not be confused.

Machine accounts are requested via the web administration pages; again, go to:

`https://www.epcc.ed.ac.uk/safadmin`

and follow the instructions.

3.3 Logging in

Once you have received a notification about your machine login account being active, retrieve your password from the administrative web-site. Now you are ready to log onto the service using `ssh`:

```
ssh -l username bluegene1.epcc.ed.ac.uk
```

or

```
ssh username@bluegene1.epcc.ed.ac.uk
```

Please replace `username` with your login name.

3.4 Password changes

Once logged onto `bluegene1.epcc.ed.ac.uk` you can change your password using the UNIX command `passwd`.

3.5 System maintenance

Presently, the service can be suspended for maintenance at very short notice.

3.6 Transferring files

Use `scp` to transfer files. E.g.

```
scp filename username@bluegene1.epcc.ed.ac.uk:
```

on your *local* unix machine will transfer the file "`filename`" into your root directory on BlueSky . Obviously you have to replace `username` with your userid. Don't forget the colon behind `bluegene1.epcc.ed.ac.uk`.

3.7 Editors

The editors `vi` and `emacs` are available on the front-end.

4 User Resource Management

4.1 Diskspace

Please be careful, no backing up is done at present. The layout of the filesystem is as follows:
`/home` - Home directory space
`/work` - Quotad space on our SAN
`/pvfs` - 2TB of parallel storage - 2 servers. This is to be treated as scratch space for e.g. staging job output files. Do not store permanent data on here as the disk will be wiped regularly.
`/work0` - READ ONLY - please migrate to `/work`
`/work1` - READ ONLY - please migrate to `/work`

4.2 The BlueSky Administration Web Site

All users have a login and password on the BlueSky Administration Web Site (aka the SAF page):

`https://www.epcc.ed.ac.uk/safadmin`

Once logged into this web page, users can find out much about their usage of the BlueSky system, including:

- machine status
- project details
- detailed usage information including report generation

These features are largely self-documenting and are not described further here.

See section 3.1 and section 13.1 for more details of the web-site.

4.3 Charging

Charging on BlueSky is done on a per *node* basis, and it is up to the user whether they wish use CO or VN mode, where 1 or 2 processors per node are utilised for computation respectively.

The standard charging policy is to multiply the elapsed wall-clock time for each job by the number of nodes allocated. Note that, due to the architectute of the machine, the number of nodes allocated will be either 32, 128, 512 or 1024, and hence may differ from what is requested. Hence charging will be done according to the following table

Nodes requested	Nodes Charged
<= 32	32
33 to 128	128
129 to 512	512
513 to 1024	1024

It is therefore most economical to request either 32, 128, 512 or 1024 nodes.

Information on the charging model used is available at http://www.epcc.ed.ac.uk/ bgapps/charging_model.html

5 Compilation

5.1 Compilers

The BlueSky service uses special versions of IBM's XL compiler suite.

5.1.1 Fortran compiler

The Fortran compilers are in the directory `/opt/ibmcmp/xlf/9.1/bin/`. For F77 you want to use

```
blrts_xlf
```

for F90

```
blrts_xlf90
```

and for F95

```
blrts_xlf95
```

These will create serial codes, which are not that useful on BlueSky . We recommend using the wrapper scripts for compiling a parallel version of your code using MPI. See section 5.2 for details.

5.1.2 C compiler

The C compiler is in the directory `/opt/ibmcmp/vac/7.0/bin/`. Use

```
blrts_xlc
```

for C code. Again this will create serial code. We recommend you to use the wrapper scripts for creating an MPI executable. See section 5.2 for details.

5.1.3 C++ compiler

The C++ compiler is installed in the directory `/opt/ibmcmp/vacpp/7.0/bin/`. Its name is

```
blrts_xlC
```

This will create serial code. We recommend to use the wrapper scripts described in section 5.2 for MPI code.

5.1.4 GNU compilers

GNU compilers are available in `/usr/bin`. These are presently *unsupported*.

5.2 Compiling MPI code

For the convenient compilation of MPI codes, a number of scripts are available. These scripts are build on top of the compilers described in section 5.1 and are located in the directory `/usr/local/bin`.

5.2.1 Fortran code

There are three scripts to compile Fortran codes using MPI. They are used for F77, F90 and F95. For a basic compile use

```
mpixlf -o hello.x hello.f
mpixlf90 -qsuffix=f=f90 -o hello.x hello.f90
mpixlf95 -qsuffix=f=f95 -o hello.x hello.f95
```

This illustrates one of the "specialities" of the IBM XL Fortran compiler: it doesn't recognise `.f90` and `.f95` file suffixes. They have to be specified with the `-qsuffix` option.

The `mpixlf` scripts passes compiler options to the compiler.

5.2.2 C code

We recommend using the `mpixlc` script to compile parallel C code

```
mpixlc -o hello.x hello.c
```

The script is designed to pass compiler options to the compiler.

5.2.3 C++ code

For parallel C++ code use

```
mpixlCC -o hello.x hello.cpp
```

The compiler recognises the suffixes `.C`, `.cc`, `.cpp` and `.cxx` as C++ source files.

Due to C++ namespace conflicts it is essential to include `mpi.h` *before* either of `iostream.h` or `stdio.h`. If you include the headers the wrong way round you will get an error of the form:
`"/bgl/BlueLight/ppcfloor/bglsys/include/mpicxx.h", line 26.2: 1540-0859 (S) #error directive: "SEEK_SET is #defined but must not be for the C++ binding of MPI".`

5.3 Useful compiler options

Still to come! But have a look in section 11.1 for advanced switches

5.4 Example Makefiles

A couple of sample makefiles are shown below. They use a basic optimisation level. For production code you might want to play with the options for more agressive optimisation.

5.4.1 MPI fixed format Fortran code

Example Makefile (`Makefile.MPI.f`) to compile code with two MPI fixed format Fortran source files: `mpihello.f` and `hello.f`.

```
MF=     Makefile.MPI.f

FC=     mpixlf
```

```
FFLAGS= -O -qarch=440 -qtune=440
LFLAGS= $(FFLAGS)

EXE=    mpihello

SRC= \
        mpihello.f      \
        hello.f


#
# No need to edit below this line
#

.SUFFIXES:
.SUFFIXES: .f .o

OBJ=    $(SRC:.f=.o)

.f.o:
        $(FC) $(FFLAGS) -c $<

all:    $(EXE)

$(EXE): $(OBJ)
        $(FC) $(LFLAGS) -o $@ $(OBJ)

$(OBJ): $(MF)

tar:
        tar cvf $(EXE).tar $(MF) $(SRC)

clean:
        rm -f $(OBJ) $(EXE) core
```

5.4.2 MPI Fortran 90 code

Example Makefile (`Makefile.MPI.f90`) to compile code with two MPI Fortran 90 source files: `mpihello.f90` and `hello.f90`.

```
MF=     Makefile.MPI.f90

FC=     mpixlf90
FFLAGS= -qsuffix=f=f90 -O -qarch=440 -qtune=440

LFLAGS= $(FFLAGS)

EXE=    mpihello

SRC= \
        mpihello.f90    \
        hello.f90
```

```
#
# No need to edit below this line
#

.SUFFIXES:
.SUFFIXES: .f90 .o

OBJ=    $(SRC:.f90=.o)

.f90.o:
        $(FC) $(FFLAGS) -c $<

all:    $(EXE)

$(EXE): $(OBJ)
        $(FC) $(LFLAGS) -o $@ $(OBJ)

$(OBJ): $(MF)

tar:
        tar cvf $(EXE).tar $(MF) $(SRC)

clean:
        rm -f $(OBJ) $(EXE) core
```

5.4.3 MPI C code

Example Makefile (`Makefile.MPI.c`) to compile code with two C source files: `mpihello.c` and `hello.c`

```
MF=     Makefile.MPI.c

CC=     mpixlc
CFLAGS= -O -qarch=440 -qtune=440

LFLAGS= $(CFLAGS)

EXE=    mpihello

SRC= \
        mpihello.c      \
        hello.c

#
# No need to edit below this line
#

.SUFFIXES:
.SUFFIXES: .c .o

OBJ=    $(SRC:.c=.o)
```

```
.c.o:
        $(CC) $(CFLAGS) -c $<

all:    $(EXE)

$(EXE): $(OBJ)
        $(CC) $(LFLAGS) -o $@ $(OBJ)

$(OBJ): $(MF)

tar:
        tar cvf $(EXE).tar $(MF) $(SRC)

clean:
        rm -f $(OBJ) $(EXE) core
```

6 File Management

6.1 Use of `/tmp`

The `/tmp` area is principally intended for the temporary storage of data used by the system and system programs. Because of the size and number of jobs normally run on the BlueSky service, it is impractical to allow users create files in `/tmp` as it soon overflows. Normally, the system ensures that there is free space on `/tmp` by periodically removing files as necessary, when they are no longer in use. If `/tmp` is filling up at a dangerous rate, however, any files there may be deleted automatically at any time, in order to ensure that the system operates safely.

For this reason, users should not normally place files on `/tmp`.

6.2 Backup policies

No backups are taken at present (early access/at risk service). Please copy files away which are important to you.

7 Porting Codes

7.1 Timers

It is often useful to include time keeping routines within your code. Since BlueSky is an MPI machine, `MPI_Wtime()` should be a good choice as a timer for most applications. `MPI_Wtime()` returns a 64-bit floating point number. Below is a simple Fortran example

```
DOUBLE PRECISION :: start, end
start = MPI_Wtime()

! code to be timed

end   = MPI_Wtime()
print*,'That took ',end-start,' seconds'
```

7.2 Default sizes

7.2.1 FORTRAN codes

Warning: This section is copied from the HPCx user guide, assuming that the two versions of the IBM compilers behave the same. *This information as not been verified!*

The default sizes and available datatypes of the IBM XL Fortran compiler are as follows

Type	Length (bytes)
`character`	1
`complex`	2×4
`double complex`	2×8
`double precision`	8
`integer/logical`	4
`real`	4

NB. double complex is not standard FORTRAN.

A portable alternative is to use the Fortran 90 KIND syntax.

Type	KIND	*	Length (bytes)
`complex`	4	8	2×4
	8	16	2×8
	16	32	2×16
`integer/logical`	1	1	1
	2	2	2
	4	4	4
	8	8	8
`real`	4	4	4
	8	8	8
	16	16	16

7.2.2 C/C++ codes

Warning: This section is copied from the HPCx user guide, assuming that the two versions of the IBM compilers behave the same. *This information as not been verified!*

These are the default sizes of the standard C/C++ datatypes of the IBM C compiler

Type	Length (bytes)
`bool`	1
`char`	1
`wchar_t`	4
`short`	2
`int`	4
`long`	4
`float`	4
`double`	8
`long double`	8

7.3 MPI Datatypes

Warning: This section is copied from the HPCx user guide, assuming that the two versions of the IBM compilers behave the same. *This information as not been verified!*

The default sizes of MPI datatypes are different on the different machines:

7.3.1 Fortran

Type	Length (bytes)
`MPI_CHARACTER`	1
`MPI_COMPLEX`	2×4
`MPI_DOUBLE_COMPLEX`	2×8
`MPI_DOUBLE_PRECISION`	8
`MPI_INTEGER`	4
`MPI_LOGICAL`	4
`MPI_REAL`	4

NB: If the `-qrealsize` compiler flag is employed to increase the precision of default reals, then the interpretation of MPI_REAL is NOT changed.

MPI defines additional datatypes with explicit sizes. These can be used to provide some portability:

Type	Length (bytes)	Type	Length (bytes)
`MPI_COMPLEX8`	2×4	`MPI_LOGICAL1`	1
`MPI_COMPLEX16`	2×8	`MPI_LOGICAL2`	2
`MPI_COMPLEX32`	2×16	`MPI_LOGICAL4`	4
`MPI_INTEGER1`	1	`MPI_LOGICAL8`	8
`MPI_INTEGER2`	2	`MPI_REAL4`	4
`MPI_INTEGER4`	4	`MPI_REAL8`	8
`MPI_INTEGER8`	8	`MPI_REAL16`	16

7.3.2 C

Type	Length (bytes)	Type	Length (bytes)
`MPI_CHAR`	1	`MPI_SHORT`	2
`MPI_DOUBLE`	8	`MPI_SIGNED_CHAR`	1
`MPI_FLOAT`	4	`MPI_UNSIGNED`	4
`MPI_INT`	4	`MPI_UNSIGNED_CHAR`	1
`MPI_LONG`	4	`MPI_UNSIGNED_LONG`	4
`MPI_LONG_DOUBLE`	8	`MPI_UNSIGNED_LONG_LONG`	8
`MPI_LONG_LONG`	8	`MPI_UNSIGNED_SHORT`	2
`MPI_LONG_LONG_INT`	8	`MPI_WCHAR`	2

8 Batch Processing

BlueSky uses the `mpirun` command to run parallel jobs. Jobs on the service must use mpirun indirectly via the `loadleveler` batch processing system. Direct access to `mpirun`, and use of the `COBALT` system are no longer available.

Currently usage of the system has to be arranged among users in consultation with the BlueSky project manager[1].

For MPI jobs *Co-processor mode* and *Virtual node mode* are the key operational modes, see section 2.2 for details. You can select the best mode for your application via the `mpirun` argument `-mode` inside your loadleveler script (see example scripts in the following section).

Note that we expect to introduce accounting in the very near future. This will require additions to the below scripts.

8.1 Sample Co-processor mode job submission file

A job submission file must be used to submit a job. Here is a sample file for running in CO mode, which can be copied and customised

```
#!/bin/bash
#
#@ job_name = myjob
#@ chips = 32
#@ wall_clock_limit = 00:10:00
#@ account_no = b00
#
#@ arguments = -exe  /home/y01/alang/hello_world/mpihello.x \
       -cwd /home/y01/alang/hello_world \
       -mode CO \
       -np $(chips)
#
#@ output = $(job_name).$(schedd_host).$(jobid).out
#@ error  = $(job_name).$(schedd_host).$(jobid).err
#
#Do not alter below unless you know what you are doing
#@ bg_size = $(chips)
#@ bg_connection = PREFER_TORUS
#@ executable = /bgl/BlueLight/ppcfloor/bglsys/bin/mpirun
#@ job_type = BLUEGENE
#@ environment = $MMCS_SERVER_IP;$BACKEND_MPIRUN_PATH
#@ class = BGL
#@ queue
```

Important note: The submission filter is sensitive to white space behind the continuation character: "\", which is used for the "`arguments`" keyword in the above sample. If your script is rejected, please check that there is no white space, e.g. a "blank" behind any of the "\".

To customise, the following variables should be changed:

- `job_name` - arbitrary name for your job
- `chips` - the number of chips (equals the number of tasks in CO mode) you wish to utilise
- `wall_clock_limit` - the maximum time your job will take.
- `account_no` - a valid budget for your project.
- `arguments` - the arguments to `mpirun`

  `-exe` name of your executable including absolute path

[1]The BlueSky project manager can be contacted via the helpdesk `epcc-support@epcc.ed.ac.uk`

`-cwd` the absolute path of your current working directory

`-mode` should be CO for Co-processor mode (see next section for Virtual node mode example)

`-np` number of tasks (should be left as `$(chips)`)

For more information on mpirun arguments see section 8.5

8.2 Sample Virtual Node mode job submission file

Here is a sample script for running in VN mode

```
#!/bin/bash
#
#@ job_name = myjob
#@ chips = 32
#@ wall_clock_limit = 00:10:00
#@ account_no = b00
#
# place 2 tasks per chip
#@ tasks = ‘ expr $(chips) + $(chips) ‘
#
#@ arguments = -exe  /home/y01/alang/hello_world/mpihello.x \
       -cwd /home/y01/alang/hello_world \
       -mode VN \
       -np $(tasks)
#
#@ output = $(job_name).$(schedd_host).$(jobid).out
#@ error  = $(job_name).$(schedd_host).$(jobid).err
#
#Do not alter below unless you know what you are doing
#@ bg_size = $(chips)
#@ bg_connection = PREFER_TORUS
#@ executable = /bgl/BlueLight/ppcfloor/bglsys/bin/mpirun
#@ job_type = BLUEGENE
#@ environment = $MMCS_SERVER_IP;$BACKEND_MPIRUN_PATH
#@ class = BGL
#@ queue
```

Important note: The submission filter is sensitive to white space behind the continuation character: "\", which is used for the "`arguments`" keyword in the above sample. If your script is rejected, please check that there is no white space, e.g. a "blank" behind any of the "\".

Note that in VN mode, 2 tasks are run on each chip so the total number of tasks will be double the value given to the `chips` variable.

8.3 Submitting and monitoring your job

Important: Please keep the following in mind when choosing a value for the `chips` variable in your batch script. When you submit and run a job, LoadLeveler will allocate your job to the smallest appropriate partition. LoadLeveler currently only has 4 sizes of partition: 32, 128, 512, 1024 chips. Hence if you run a 256 chip job, it will run on only 256 chips, but LoadLeveler will reserve 512 chips and prevent other users from utilising the remaining 256 chips. Where possible please try to stick to the above sizes.

Submit the batch script with `llsubmit`:

```
alang@bglfen1$ llsubmit mpihello.ll
llsubmit: The job ‘‘bglsn.epcc.ed.ac.uk.364’’ has been submitted.
```

and check status with `llq -b`:

```
alang@bglfen1$ llq -b
Id                       Owner      Submitted   LL BG PT Partition        Size
_________________________ __________ ___________ __ __ __ _________________ _______
bglsn.364.0              alang       1/6  14:10 R  ST RE RMP06Ja110522265 32

1 job step(s) in queue, 0 waiting, 0 pending, 1 running, 0 held, 0
preempted
```

To cancel a job type the command

```
llcancel <job_id>
```

where `<job_id>` is given in the `Id` field above.

The files `myjob.bglsn.<job_id>.out` and `myjob.bglsn.<job_id>.err` which contain standard output and error respectively will be created in the working directory when the job is finished.

8.4 Multistep Submission

It is often desirable to, as part of your job, perform additional operations besides running the parallel code. For example, one might wish to prepare data beforehand or perform some manipulation of the results afterwards. Loadleveler allows you to perform multiple steps within the one submission script, as is demonstrated by the below simple example script which performs two steps.

```
# This job command file lists two job steps called "step1"
# and "step2". "step2" only runs if "step1" completes
# with exit status = 0. Each job step requires a new
# queue statement.
#
# @ account_no = b00-apps
# @ step_name = step1
# @ executable = executable1
# @ output = step1.out1
# @ error = step2.err1
# @ queue
# @ dependency = (step1 == 0)
# @ step_name = step2
# @ executable = executable2
# @ output = step2.out1
# @ error = step2.err1
# @ queue
```

For each step, `executable` should be set to a program or script which performs the required operations.

For a real run, one of these steps should resemble one of the sample scripts given in the previous sections, i.e. `executable` would be set to `mpirun` and the other relevant keywords would be added. Here, however, note that the `job_name` keyword has been replaced by a `step_name` keyword for each step. Also, the `dependency` keywords ensure that if a step fails the job will be cancelled before the next step.

For each step, a seperate entry `bglsn.<job_no>.<i>` will appear in the output of `llq`, where the integer `i` identifies the step.

8.5 More on mpirun

Arguments available to mpirun include

- `-np` specifies the number of MPI tasks.
- `-mode` select Co-processor mode (CO) or virtual node mode (VN). See section 8.5.1 for further details.
- `-verbose` controls the verbal diarrhoea produced by `mpirun`. Hint: you want to set this to 0, which is not the default.
- `-exe` specifies the executable. This needs to be the **full** path.

8.5.1 Running in Virtual-node mode or Co-processor mode

Each chip has two processors. As explained section 2.2, it is possible to place one or two MPI-tasks on each chip. When placing one task on each chip, the second processor of each chip is used as a communication co-processor, hence this is called Co-processor mode. Placing two task on each chip is called Virtual-node mode. In Virtual-node mode the amount of memory per task halfes.

Use the `-mode` option of `mpirun` to specify whether your application should run in Co-processor (`-mode CO`) or Virtual-node mode (`-mode VN`). To determine which mode is the more advantageous mode for your application, you should keep in mind that in virtual node mode you can place twice the number of tasks (`mpirun`-option `-np`) for the same hardware cost. For example you should be comparing 32 tasks in Co-processor mode with 64 tasks in Virtual-node mode. The mode giving the faster timings is "*your mode*".

8.5.2 Process Mapping

When running parallel codes using `mpirun`, the MPI tasks are mapped to the processors on the machine. The performance of the code can depend on the mapping used, and communication times can be minimised with mappings which resemble the domain of the physical problem being computed.

For information on default and customised process mapping, see "Process Mapping on EPCC's BlueGene/L Service" at
`http://www.epcc.ed.ac.uk/ bgapps/restricted/UserGuide/documents/mapping.pdf`.

8.5.3 Output from your job

If you do not want to include the full path of your output file(s) into your code, you can use the `-cwd` option of mpirun to specify the directory in which your output files get written. This option takes a single argument, which has to be present.

8.5.4 Environment Variables

In order for a variable to be in the environment of the running code, you must use the `-env` argument to `mpirun`. e.g. if you want to set `VARIABLE1=VALUE1` and `VARIABLE2=VALUE2` you would include the argument

```
-env "VARIABLE1=VALUE1 VARIABLE2=VALUE2"
```

The double quotations are necessary, and you can set as many variables as required.

8.5.5 Further options of `mpirun`

To enquire about further options of `mpirun` use

```
mpirun -h
```

8.6 Accounts linked to more than one project

A number of users have accounts, which are linked to more than a one project. On BlueSky we implement projects via unix groups. Use the Unix command `groups` to learn about the projects an account is associated with. For example

```
username@bglfen1 ~: groups
b00 y01 b18
```

shows the user's account to be associated with the projects `b00`, `y01` and `b18`. If this user needs to run a job accounted to `b18`, this needs to become his *first* Unix group. Use the `newgrp`-command before job submission to achieve this

```
username@bglfen1 ~: newgrp b18
username@bglfen1 ~: groups
b18 y01 b00
```

This will be temporarily and needs to be repeated every time you log on.

9 Tools

9.1 Debugging with gdb

The Gnu Debugger `gdb` can be used to interactively debug your parallel code. You must be at your terminal when the job runs.

In your loadleveler job submission file, change the line

```
#@ executable = /bgl/BlueLight/ppcfloor/bglsys/bin/mpirun
```

to

```
#@ executable = /bgl/local/bin/mpirun_gdb
```

and submit as usual.

When the job begins to run, as usual the file `<job_name>.bglsn.<job_no>.out` will be created in your working dir (where `<job_name>` is specified in the loadleveler job submission file and `<job_no>` is a unique number given to the job). Type the command

```
tail -f <job_name>.bglsn.<job_no>.out
```

to monitor this file. After a short while, a list of IP addresses and ports will be written to this file; each entry corresponding to a MPI task, e.g.

```
MPI Rank 0: Connect to 172.30.1.110:7300
...
```

Enter `Ctrl-c` to exit the `tail` command.

You now have 2 minutes to connect to the node corresponding to the MPI task which you wish to debug:

- Make sure you are in the directory containing the executable and type

  `/bgl/local/packages/gdb-6.5/bin/gdb <executable>`

  where `<executable>` is the name of your executable.

- From within the gdb program, you can then connect to the process you wish to debug by typing

  `target remote <IPaddress:port>`

  e.g.

  ```
  (gdb) target remote 172.30.1.110:7300
  Remote debugging using 172.30.1.110:7300
  ```

  Now wait, and within 2 minutes you will get a `(gdb)` prompt (ignore any error message regarding the file `start.c`).

- You are now ready to use `gdb` as normal, e.g. set breakpoints using `break <sourcecode:line_number>` (or use a function name instead of a line number), run the code up to each breakpoint using the `c` (continue) command and print the value of the variables at these points using the `print` command:

  ```
  (gdb) break mpihello.c:15
  Breakpoint 2 at 0x100558: file mpihello.c, line 15.
  (gdb) c
  Continuing.

  Breakpoint 2, main (argc=1, argv=0x1feaa9c0) at mpihello.c:15
  15        MPI_Pcontrol(1);
  (gdb) print rank
  $1 = 0
  (gdb) print size
  $2 = 4
  ```

- When finished, to exit gdb type `q` and confirm with `y`. If the MPI program has not finished it will be killed.

For more information on `gdb` commands, type

`man gdb`

9.2 Traceback

When debugging a code which crashes, `traceback` can be used to determine which source code line is crashing the program.

To use:

- Compile your code with the `-g` compiler option.
- Run your code using `mpirun` as normal.
- For the task which the code is crashing on, a file `core.<taskid>` will be written to the working directory. Look in this file for the line directly below the line

  `Function call chain:`

- Then if this line reads, for example,

`0x00100560`

type the command

`addr2line -e helloworld.x 0x00100560`

where `helloworld.x` is the executable.

The output

`/home/y01/alang/code/helloworld/helloworld.f90:10`

shows you the code crashed at line 10 of `helloworld.f90`.

9.3 MPI trace

`mpitrace` is a profiling and tracing tool which allows you to analyse the MPI communication events within your program, giving an overview of the makeup of your code and potentially an insight into any communication bottlenecks.

To use:

- Link the application code with

  ```
  -L/bgl/local/lib/mpitrace -lmpitrace_f (Fortran)
  -L/bgl/local/lib/mpitrace -lmpitrace_c (C)
  ```

  If using a mixed language C and Fortran application then use `mpitrace_c`.

- Run the application

Files will be created in the working directory with names `mpi_profile.<taskid>` for task zero and the tasks which had the maximum, minimum and median communication time. Each of these files contains information including the number of calls to, the time taken by and the average message length passed to each MPI routine. Also given are total communication and total elapsed times. The `mpi_profile.0` file contains an extra communication time summary for all tasks. This summary is given in terms of the topology coordinates of the tasks as well as the task IDs, e.g.

```
MPI tasks sorted by communication time:
taskid  xcoord  ycoord  zcoord  procid    total_comm(sec)
     0       0       0       0       0         0.075
     1       1       0       0       0         0.199
     4       0       1       0       0         0.311
     6       2       1       0       0         0.312
     2       2       0       0       0         0.312
     5       1       1       0       0         0.312
     3       3       0       0       0         0.312
     7       3       1       0       0         0.312
```

This potentially provides a clearer picture of how communication bottlenecks are related to the design of the application.

9.3.1 Function Trace

The `function_trace` feature of `mpitrace` can be used to gain information about the times taken by functions or subroutines within the code. This is useful in determining the most computationally expensive sections of applications. To use, link to the `mpitrace` library as above but include the compilation option

```
-qdebug=function_trace
```

and run with the mpirun option

```
-env "FLAT_PROFILE=yes"
```

The files `flat_profile.<taskid>` (analogous to `mpi_profile.<taskid>`) are then created. These list the times taken by the functions inclusively (where calls to other functions are included in the times) and exclusively. A percentage breakdown of the total execution time is also given, e.g.

```
-----------------------------------------------------------------------
Elapsed-time flat profile for MPI task 0.
-----------------------------------------------------------------------
Self(seconds) does not include calls to other routines, Inclusive does.
-----------------------------------------------------------------------
  %Time  Self(seconds)    Inclusive       #Calls  Routine
  27.63        0.2348        0.2348            1  pgmwrite
  14.94        0.1269        0.1269            1  datread
   0.39        0.0034        0.2381            1  __image_mpi_comm_NMOD_write_picture
   0.21        0.0018        0.1287            1  __image_mpi_comm_NMOD_read_edges
   0.15        0.0013        0.0013            1  __image_mpi_comm_NMOD_make_cart_world
   0.13        0.0011        0.0011            1  __image_work_NMOD_print_times
   0.02        0.0002        0.1289            1  __image_work_NMOD_edge_init
```

9.4 MP_Profiler

9.4.1 Profiling

`MP_Profiler` is essentially a more advanced version of `mpitrace`. Communication statistics are viewed with the `peekperf` utility, making it easier to relate to your code and help find any bottlenecks.

For detailed information, please see the documentation at

`/bgl/local/packages/hpct_bgl/doc/mp_profiler/mpt_manual.pdf`

To use, when building your Fortran application link with

`-L/bgl/local/packages/hpct_bgl/lib -lmpitrace_f -llicense`

For C, replace `-lmpitrace_f` with `-lmpitrace_c`

Run your code as normal and The files `mpi_profile_<taskid>.viz`, for each mpitask, will be written to the working directory. View these with the command

`/bgl/local/packages/hpct_bgl/bin/peekperf *.viz`

A `peekperf` window will open, with a list of the procedures which contain communications on the left and the corresponding source code on the right. Expanding each procedure shows the MPI communications contained within. Right clicking on the MPI call name brings up a list timing statistics for task zero and the tasks which had the maximum, minimum and median communication time.

Also written to the working directory are `mpi_profile.<task>` text files, similar to those described for mpitrace above.

9.4.2 Tracing

Viewing a trace of the code allows you to visualise all the communications which occur. By default, a trace file `single_trace` which contains information for task zero and the tasks which had the maximum, minimum and median communication time is created, and can be viewed with the command

```
/bgl/local/packages/hpct_bgl/bin/peekview single_trace
```

9.5 TAU

The `TAU` utility is a more sophisticated profiling and tracing alternative to `mpitrace` and `function_trace`. It can be used to gain and graphically display information on the performance of procedures and communications within your code.

To use, include the lines

```
TAUROOTDIR = /bgl/local/packages/tau
include $(TAUROOTDIR)/bgl/lib/Makefile.tau-mpi-pdt
```

at the start of your makefile, and preceed your compiler with `$(TAU_COMPILER)`, e.g.

```
FC = $(TAU_COMPILER) mpixlf90
```

Then running the job as normal will result in the files `profile.<taskid>.0.0`, for each MPI task, being created in the working directory.

To view these, add `/bgl/local/packages/tau/ppc64/bin` to your PATH and then type the command `paraprof` whilst in the directory containg the `profile.<taskid>.0.0` files. A window will appear displaying a horizontal bargraph of the relative time spent in each function for each MPI task, and for an average over all tasks. Further information is provided via left and right clicking on different parts of the graph and/or using the drop down menus. For example:

- To see a textual profile for MPI task 0, click right mouse button over `n,c,t 0,0,0` and select `Show Statistics Window`. The resulting list can then be sorted in different ways via the drop down `Options` menu.
- To see the relative function profile for MPI task 0 click the left mouse button on `n,c,t 0,0,0` (in the main bargraph window).
- To see the profile of a specific function over all nodes, click the left button on that function.
- To change the colour scheme, click the right button over the one of the bars and select `Assign Function Color`.

For a more detailed guide on how to use `TAU` on this machine, including the generation of trace files which can be viewed by external utilities, see "TAU on EPCC's BlueGene/L Service" at http://www.epcc.ed.ac.uk/ bgapps/restricted/UserGuide/documents/tau.pdf.

9.6 KOJAK

`KOJAK` is a set of tools designed to analyse the performance of and find bottlenecks in parallel applications. For full information, please see the documentation and examples in the `/bgl/local/packages/kojak/` directory and the web page http://icl.cs.utk.edu/kojak/.

To use on BlueSky, first a few directives must be added to the source code. At least
`!POMP$ INST INIT`
`!POMP$ INST BEGIN(name)`
for Fortran, or
`#pragma pomp inst init`
`#pragma pomp inst begin(name)`
for C are needed as the first executable statements of the main routine and
`!POMP$ INST END(name)` (Fortran)
`#pragma pomp inst end(name)` (C)
are needed as the last executable statements. `name` can be chosen as the name of the main routine.

Additionally, the user can choose to instrument any additional subroutines or segments of code using the 'begin' and 'end' statements above to gain more clarity (about how events relate to the program structure) at the analysis stage.

Then, add the location of the relevant executables to your path with

```
export PATH=$PATH:/bgl/local/packages/kojak/bin
```

The code application should be built with the name of the compiler preceded by the `kinst-pomp` command, e.g.

```
kinst-pomp mpixlf90 -qsuffix=f=f90 source_code.f90 -o executable.x
```

The executable can then be run in the normal way resulting in the trace files `a.<taskid>` being created in the working directory for each MPI task. These should then be merged with the command

```
elg_merge <N> a
```

where `<N>` is the number of MPI tasks.

This merged trace can then be automatically analysed with the command

```
expert a.elg
```

resulting in the creation of the file `a.cube` which can be browsed with the `cube` viewer:

```
cube a.cube
```

The `cube` utility is designed to make it easy to determine performance properties of the application via hierarchical (performance, call and system) trees. A colour coding system helps to identify hot-spots. The manual can be downloaded from http://icl.cs.utk.edu/kojak/cube.

9.6.1 Hardware Counters with KOJAK

The above use of `KOJAK` will provide information about the communications within the application. Information on the use of hardware resources can also be obtained.

Prior to running the job, the user must specify which hardware counters to invoke via the environment variable `ELG_METRICS` by specifying them as a colon separated list. For example, specifying the `mpirun` option

```
-env ELG_METRICS=PAPI_TOT_CYC:PAPI_FMA_INS
```

will provide information on Total Cycles and Floating Multiply Add instructions completed.

The following counters are available:

```
PAPI_L3_TCM  -  Level 3 total cache misses
PAPI_L3_LDM  -  Level 3 load misses
PAPI_L3_STM  -  Level 3 store misses
PAPI_FMA_INS  -  Floating Multiply Add instructions completed
PAPI_TOT_CYC  -  Total cycles
PAPI_L2_DCH  -  L2 data cache hits
PAPI_L2_DCA  -  L2 data cache accesses
PAPI_L3_TCH  -  L3 total cache hits
PAPI_FML_INS  -  Floating Multiply instructions
PAPI_FAD_INS  -  Floating Add instructions
```

In addition, there are another 5 derived events unique to the BG/L architecture:

```
PAPI_BGL_OED     - FPU Oedipus operations
PAPI_BGL_TS_32B  - 32B torus packets sent in any direction
```

```
PAPI_BGL_TS_FULL - UPC cycles any link has no tokens
PAPI_BGL_TR_DPKT - No data packets sent on any tree channel
PAPI_BGL_TR_FULL - UPC cycles any tree receiver is full
```

Note that not all counters are compatible with one another so seperate runs may be needed.

9.7 Xprofiler

`Xprofiler` is a straightforward profiling tool with an intuitative graphical interface which helps you identify the computationally intensive parts of your code.

To use, compile your code with the compilation options

```
-pg -g
```

and run as normal. The files `gmon.out.<taskid>`, for each MPI task, will be created in the working directory. Each of these can be viewed individually with the command

```
/bgl/local/packages/hpct_bgl/bin/Xprofiler <executable> gmon.out.<taskid>
```

The display will graphically show the time taken by procedures in the code. For more clarity, first select Filter->Uncluster Functions. Then select View->Zoom in and click on the region of the graph of interest. Shown are rectangles representing procedures, the width of the which represents the inclusive time spent in the procedure and the height represents the exclusive time: the largest shapes contain the most computationally expensive parts of the code. Right clicking on the boxes gives the option to view the associated source or disassembler code.

9.8 HPM

The Hardware Performance Monitor (HPM) tool provides information about the hardware events which occur when your program runs, which can be useful in determining how your code is performing on the specific architecture of the machine.

For a fortran code, insert the header

```
#include "f_hpm.h"
```

and insert the lines

```
call f_hpminit(1,'<executable_name>')
```

and

```
call f_hpmterminate(1)
```

immediately after MPI initialisation and immediately before MPI finalisation respectively. `<executable_name>` should be replaced by the name of your executable.

Each section of code to be measured should be sandwiched between calls to

```
call f_hpmstart(<n>,'<label>')
```

and

```
call f_hpmstop(<n>)
```

where `<n>` is a integer between 0 and 100, unique for each code section, and `<label>` should be chosen to identify each section.

The equivalent C statements are

```
#include <libhpm.h>
```

```
hpmInit(1,"<executable_name>");
hpmTerminate(1);

hpmStart(<n>,"<label>");
hpmStop(<n>);
```

When building your program specify the compilation options

```
-g -I/bgl/local/packages/hpct_bgl/include
```

For fortran programs, also specify the compilation option `-qsuffix=cpp=<suffix>` where `<suffix>` is the suffix of your source code filename, likely to be `f` or `f90`.

Link the code with

```
-L/bgl/local/packages/hpct_bgl/lib -lhpm.rts -lbgl_perfctr.rts
```

and run with the `mpirun` option

```
-env HPM_EVENT_SET=<event_set>
```

where `<event_set>`, an integer in the range of 0 to 16, specifies the event set which is to be measured. For details of which counters are included in each set, see the file

```
/bgl/local/packages/hpct_bgl/doc/hpm/event_sets.txt
```

The default is event set 0.

The measurement result files `perfhpm00001.<taskid>` and `hpm00001_<executable_name>_0000<taskid>.viz` for each MPI task will be created in the working directory. The former are plain text files. The latter can be viewed using `peekperf`:

```
/bgl/local/packages/hpct_bgl/bin/peekperf *.viz
```

A window will open which contains on the left each of your measured code section labels and on the right the corresponding source code. To view the measurement results, right click on the code section label. A new window will open, in which the the desired metric result can be displayed via the `Metric Options` drop down menu.

9.9 Paraver

Paraver is a comprehensive performance analysis and visualisation tool. It can be used to generate and view traces of your parallel code.

For C, Link your code with

```
-L/bgl/local/packages/paraver/lib -lmpitrace.rts \
 -L/bgl/local/packages/papi-2.3.4.3.bgl/lib -lpapi.rts \
 -L/bgl/BlueLight/ppcfloor/bglsys/lib -lbgl_perfctr.rts -lmpich.rts \
 -lmsglayer.rts -ldevices.rts -lrts.rts
```

For Fortran, link as above but with `-lmpitrace.rts` replaced with `-lmpitracef.rts`.

Then run with the `mpirun` argument

```
-env "MPITRACE_ON=1 MPTRACE_BUFFER_SIZE=100000 \
     MPITRACE_MPI_COUNTERS_ON=1"
```

During the run, Paraver will have created a temporary trace file for each process (`*.mpit` and `*.sim` files).

In order to view the results, first add the Paraver directory to your path with

```
export PARAVER_HOME=/bgl/local/packages/paraver
```

Then, add the location of the relevant executables to your path with

```
export PATH=$PATH:$PARAVER_HOME/bin
```

Now submit an instruction to pack the individual profile files into one global output. This is undertaken by issuing the command:

```
mpi2prv *.mpit -s *.sym -o trace_prm.prv
```

To view the resulting tracefile use the command:

```
paraver trace_prm.prv
```

Full details on how to utilise the wide range of functionality offered by paraver are available at http://www.cepba.upc.edu/paraver .

For example, to see a trace of MPI events

- Select `Configuration->Load Windows->OMPITrace->MPI->views->MPI_call.cfg`
- Optional: make the resuting window larger by dragging the corner, then right clicking on the graph and selecting `Scale->Fit X-Scale`
- Get a colour key to the graph by right clicking the graph and selecting `Show Properties`
- Then you can zoom in on any section by right clicking on the graph and selecting `Zoom`, then choosing the area of interest by left clicking.

To see a table of MPI statistics,

- Select `Configuration->Load Windows->OMPITrace->MPI->analysis/mpi_stats.cfg`

10 Libraries

A number of libraries are installed on the system. Most libraries are installed in

```
/bgl/local/lib
```

or a subdirectory of this.

10.1 BLAS Library

The BLAS (Basic Linear Algebra Subprograms) are the basic building blocks for many applications using linear algebra. There are presently two versions of the BLAS Library installed on BlueSky . You should benchmark your application, to determine which version performs best for your needs.

The archive `libblas.a` has been compiled directly from the F77 source downloaded from `http://www.netlib.org/blas/`. To link against this library add

```
-L/bgl/local/lib/ -lblas
```

at the link step. We presently provide a second version of the BLAS, build for Argonne Lab. To link this version add

```
-L/bgl/local/lib/ -lblas440
```

at the link step. We expect a machine optimised version of the BLAS library soon.

10.2 FFTW 2.1.5

Presently we have two versions of FFTW 2.1.5 installed on the system. One version has been specifically optimsed for the double FPU, the other version is the standard FFTW obtaind from `http://www.fftw.org`.

10.2.1 Optimised FFTW

The optimised library uses 8-byte floats and 16-byte complex numbers – sorry no single precision version at present. It is installed in

```
/bgl/local/lib/FFTW2_1_5
```

To build your executable you should add

```
-L/bgl/local/lib/FFTW2_1_5/lib -ldfftw -ldrfftw
```

at the link step. The include files live in `/bgl/local/lib/FFTW2_1_5/include/`.

This is the FFTW-GEL library from the Technical University of Vienna compiled using IBM's xlc8.0/xlf10.1 compiler. See `http://www.complang.tuwien.ac.at/skral/fftwgel.html` for further details.

10.2.2 Standard FFTW

The standard FFTW 2.1.5 has been installed for both single, double precision (real and complex). MPI versions are also included. It is installed in

```
/bgl/local/lib/fftw
```

To build your executable you should add (e.g for single precision)

```
-L/bgl/local/lib/fftw/lib -lsfftw -lrfftw
```

at the linking stage. Double precision libraries have are denoted by a "d". MPI libraries include are denoted by "_mpi".

The include files live in `/bgl/local/lib/fftw/include/`. Some information files live in `/bgl/local/lib/fftw/info/`.

10.3 Global Arrays

The Global Array (GA) toolkit allows the use of "Global Array" data structures which are distributed across nodes but can be treated as if they are stored in shared memory. For more information see http://www.emsl.pnl.gov/docs/global/user.html. Global Arrays require the use of the Aggregate Remote Memory Copy (ARMCI) library, which is also available.

To build ARMCI/GA applications on BlueSky, several compiler flags and libraries are required. When compiling, the include paths

```
-I /bgl/BlueLight/ppcfloor/bglsys/include/ga
```

and/or

```
-I /bgl/BlueLight/ppcfloor/bglsys/include
```

will be required, depending on which header files you have included in your source code.

You should link your code with the following:

```
-L /bgl/BlueLight/ppcfloor/bglsys/lib
```

plus whichever of

```
-larmci -lglobal -llinalg -lma -lpario -ltcgmsg-mpi
```

your application requires, plus

```
-lfmpich_.rts -lmpich.rts lmsglayer.rts -lrts.rts -ldevices.rt
```

Some applications that use both Fortran and C might require `-lfmpich.rts` (note: No underscore) in addition to the libraries listed.

An example compile might look like:

```
blrts_xlc -c my_GA_prog.c -DBGL -DBLRTS - DBGML -DMPI \
-I /bgl/BlueLight/ppcfloor/bglsys/include
blrts_xlc -o my_GA_prog.exe my_GA_prog.o \
-L /bgl/BlueLight/ppcfloor/bglsys/lib -lglobal -lma \
-llinalg -larmci -lfmpich_.rts -lmpich.rts -lmsglayer.rts \
-lrts.rts -ldevices.rts
```

10.4 ESSL

The ESSL (Engineering and Scientific Subroutine Library) contains a large number of highly tuned serial numerical routines. It includes the BLAS routines for basic linear algebra, and also routines that cover areas such as linear equation solving, both dense and sparse eigensolvers, fourier analysis, quadrature, interpolation, random number generation and sorting. Version 4.2.2 of ESSL is installed on Blue Gene. A PDF document detailing the full capabilities of ESSL and the interfaces to the routines is available here: `http://publib.boulder.ibm.com/epubs/pdf/am501403.pdf`

The ESSL library is installed under the path

```
/opt/ibmmath/essl/4.2
```

To build a Fortran executable using ESSL you need to link with

```
-L/opt/ibmmath/lib -lesslbg
```

To build a C/C++ executable you need to compile with the appropriate C/C++ compiler and then link with the Fortran compiler as ESSL has been built with the Fortran compiler. E.g. for a C code use

```
blrts_xlc -c mycode.c -I/opt/ibmmath/include
blrts_xlf90 mycode.o -L/opt/ibmmath/lib -lesslbg
```

11 Optimisation

11.1 Compiler flags

The use of appropriate optimisation switches at compile time can lead to significant (sometimes dramatic) improvements in execution time

Fortran and C compilers have a similar set of optimisation switches, some of these are described briefly here. Users should refer to the relevant compiler documentation for more detailed information.

- `-qarch=440` and `-qarch=440d`

  The `-qarch` flag specifies the instruction set architecture of the machine, and may take advantage of instructions only available on the specified machine. The flag `-qarch=440` specifies a

PowerPC 440 core with standard PowerPC floating-point unit. When specifying `-qarch=440d` the compiler will attempt to generate code which exploits the double FPU, see section 2.1 for details on the FPU. To exploit the double FPU, the compiler might need help with respect to 16-byte alignment, see section 11.2.

You have to test your code. It is not clear that `-qarch=440d` will generate the faster code compared to `-qarch=440`.

- `-qhot`

  The `-qhot` flag forces the compiler to carry out certain high-order transformations of the source code. For example, loops with similar trip counts and no dependencies may be merged; inner and outer loops may be interchanged so that the innermost loop counter varies most rapidly; in some cases, intrinsic functions will be extracted from loops and computed in batches (vectorisation). `-qhot` is turned on automatically at `-O4` and higher.

- `-qtune=440`

  The `-qtune` flag biases the optimisation towards execution on a given machine, `-qtune=440` specifies that this is for a PowerPC 440 processor system.

- `-O0`, `-O2`, `-O3`, `-O4`, `-O5`

  The `-O` flag is the main compiler optimisation flag and can be specified with a range of values.

  - `-O0` (Fortran only)
    Fast compilation and full debugging support.
  - `-O2`
    Comprehensive low-level optimisation.
    Global assignment of user variables.
    Elimination of redundant or unused code.
    Machine specific scheduling instructions used..
  - `-O3`
    Source manipulation where indicated.
    Inner loop unrolling.
    Software pipelining.
    Whole procedure scoping.
  - `-O4`
    High order transformations (see also `-qhot`).
    Inter-procedural analysis (IPA) & inlining.
    Automatic architecture detection & cache tuning.
  - `-O5`
    Aggressive IPA.
    Inlining.

11.2 16-byte alignment of data

The design of the floating point units of the processing cores differ substantially from the ones presently used in other architectures. In order to make efficient use of the, the compiler might need help. This section describes this in some detail.

The floating point units have 16-byte wide registers, which can hold two 8-byte floating point words[2]. The 8 leading bytes are called *primary part* of the register, the other 8 bytes are the *secondary part* of the register. The floating point unit offers instructions which operate simultaneously on both parts of the

[2] 16-byte arithmetic is not supported by these units

registers involved. A simple example is the simultaneous execution of two additions. Let a_p denote the primary part of register a and a_s its secondary part. The following instructions can be simultaneously executed in a SIMD-like fashion:

$$t_p = a_p * b_p + c_p$$
$$t_s = a_s * b_s + c_s$$

There are a number of special instructions which act differently on the two parts and even mix primary and secondary parts, for example:

$$t_p = -a_s * b_s + c_p$$
$$t_s = a_s * b_p + c_s$$

For this to work efficiently, the FPU has to be able to load two 8-byte floating point numbers from the Level 1 cache in a single instruction. The first of these numbers goes into the primary part of a register and second number goes into the secondary part of the same register. This is know as a *quadword load*.

The Level 1 cache has a 32-byte cache line. A quadword load can only be issued if the operands to be loaded are consecutive and occupy either the first or the second half of the cache line. This is referred to as 16-byte alignment. In order to generate code, which uses both sides of the floating point unit, the compiler has to know about the alignment properties of your data. Two cases have to be distinguished:

- **The data in question is local to the compilation unit**
  If the data array is local to the present compilation unit (e.g. static array in the present subroutine) the compiler will align the array for you. The first element of each array will be placed directly after a 16-byte boundary.

  Fortran case: If `a` is declared as

  ```
  Real(kind(1.0d0)), dimension(1:N) :: a
  ```

  The element `a(1)` will be allocated after the boundary. Assuming you want access the array `a` in a loop with a stride of one. The compiler can instruct `a(1)` and `a(2)` to be simulatenously loaded into a single 16-byte register. The next load will simultaneously fetch `a(3)` and `a(4)` and so on. Assuming that you use the right compiler flags (`-O3 -qarch=440d -qtune=440` or higher), the compiler doesn't need any further help in this case and will utilise both sides of the FPU. If you declare `b` as

  ```
  Real(kind(1.0d0)), dimension(2:N) :: b
  ```

  `b(2)` will be placed after the 16-byte boundary.

  C case: This is similar to the Fortran case, but with the index shifted by 1. If `a` is declared as

  ```
  double a[N];
  ```

  The element `a[0]` will be allocated after the boundary. The compiler can instruct `a[0]` and `a[1]` to be simulatenously loaded into a single 16-byte register. The next load will fetch `a[2]` and `a[3]`. Again, assuming that you use the right compiler flags (`-O3 -qarch=440d -qtune=440` or higher), the compiler doesn't need any further help in this case and will utilise both sides of the FPU.

- **The data is *not* local to the compilation unit**
  If the data is not local to the present compilation unit (e.g. handed in as a dummy argument of the present subroutine), the alignment properties are not known at compile time. In the case the alignment properties need to be declared to the compiler. With the `alignx` function you can assert to the compiler which array element is at the 16-byte boundary. The syntax is slightly different in C and Fortran.

  C usage: Let us assume, the variables `a`, `b` and `c` have been declared as

  ```
  double *a, *b, *c
  ```

in another unit of your code and are passed to the present routine. For the double FPU to work, the objects these variables are pointing to have to be disjoint and the compiler needs to be told that is the case in a `pragma`. C-code summing the vectors `b` and `c` into `a` should look like

```
#pragma disjoint(*a, *b, *c)
__alignx(16,a);
__alignx(16,b);
__alignx(16,c);
for (i=0, i<n, i++)
  {a[i] = b[i] + c[i];}
```

The above calls to `__alignx` assert to the compiler `a`, `b` and `c` are pointing to an odd entry of the array, which are 16-byte aligned. It is to the programmer to ensure this is indeed the case. If you get this wrong, the performance penalty will be extremely high.

Fortran usage: In Fortran the situation is slightly easier, since different variables passed into a subroutine have to be disjoint as part of the Fortran standard, so everything can be assumed as disjoint. Let us assume `a`, `b` and `c` have been declared in another program unit as

```
Real(kind(1.0d0)), dimension(1:N) :: a, b, c
```

and are passed into the present unit. If they are declared as above, it should be safe to assume that `a(1)`, `b(1)` and `c(1)` sit right behind the 16-byte boundary. Use the function `alignx` to declare the alignment properties of the arrays inside your subroutine/function:

```
call alignx(16,a(1))
call alignx(16,b(1))
call alignx(16,c(1))
do i=1, N
  a(i) = b(i) + c(i)
end do
```

Care has to be taken when passing sub-arrays into subroutines. It should be save if you pass a sub-array starting with an odd entry. Obviously if you start with an even entry, your `alignx` asserts the wrong thing to the compiler and the performance of your code will be extremely poor.

Again you should use the compiler flags `-O3 -qarch=440d -qtune=440` or higher when compiling your code.

12 Applications

Still to come!

13 Support, Training and Documentation

13.1 The BlueSky Administration Web Site

The BlueSky Administrative Web Site can be accessed from
`https://www.epcc.ed.ac.uk/safadmin`

13.2 The Helpdesk

Comments, Questions and Queries about the BlueSky service should be directed to the EPCC Helpdesk `epcc-support@ed.ac.uk`

13.3 Training

Please check `http://www.epcc.ed.ac.uk/computing/training/` for details on forthcoming EPCC training courses.

13.4 Documentation

See section 14 for pointers.

14 References and Further Reading

14.1 BlueSky service documentation:

This document is available from the BlueSky website.

14.2 IBM Product Documentation:

There are presently two IBM Redbooks on BlueGene available. The IBM Redbook: "*Unfolding the IBM eServer Blue Gene Solution*" offering detailed information on the BlueGene system is available from:
`http://www.redbooks.ibm.com/abstracts/sg246686.html?Open`.
The IBM Redbook: "*BlueGene/L: Applications Development*", which is more focused on the programming environment, is available `http://www.redbooks.ibm.com/abstracts/sg247179.html?Open`.

More detailed documentation on LoadLeveler is available from `http://publib.boulder.ibm.com/epubs/pdf/am2ug303.pdf`.

14.3 Useful papers and slides (IBM):

1. "*An Overview of the BlueGene/L Supercomputer*"
   NR Adiga, et al.,
   `http://sc-2002.org/paperpdfs/pap.pap207.pdf`
   Paper presented at SC2002. Detailed overview on the BG/L hardware.
2. "*Unlocking the Performance of the BlueGene/L Supercomputer*"
   George Almasi, et al.,
   `http://www.sc-conference.org/sc2004/schedule/pdfs/pap220.pdf`
   Paper presented at SC2004 in Pittsburgh. Overview on the BG/L architecture, porting issues, application performance.
3. "*A high performance SIMD Floating Point Unit for the BlueGene/L: Architecture, Compilation and Algorithm Design*"
   Leonardo Bachega, et al.,

`http://www.research.ibm.com/people/g/gupta/pact04.pdf`
Preprint (2004). Discussion of dual FPU and explanations on what the compiler can do for you to exploit it.

4. BlueGene/L Workshop in Groningen:
`http://www.lofar.org/BlueGene/`

   Relevant for code porting issues:
   "*Porting Applications to the Blue Gene/L Platform*"
   Gyan Bhanot, based on Bob Walkup's materials
   `http://www.lofar.org/BlueGene/Bhanot_Day02.pdf`

5. "IBM Journal of Research and Development"
   Special edition on Blue Gene, Vol 49, No 2/3, 2005
   `http://www.research.ibm.com/journal/rd49-23.html`

6. "*A Tutorial on BG/L Dual FPU SIMDization*"
   Peng Wu, Rohini Nair and Alexandre Eichenberger
   `http://www-unix.mcs.anl.gov/ gropp/projects/parallel/BGL/docs/tutorial-BGL-simdizatior`
   Tutorial on how to help the compiler use the Hummer[2]

14.4 Third-party Documentation